Automata for Nogood Recording in Constraint Satisfaction Problems [⋆]

Guillaume Richaud[1], Hadrien Cambazard[1], Barry O'Sullivan[2], and Narendra Jussien[1]

[1] École des Mines de Nantes – LINA CNRS FRE 2729
4 rue Alfred Kastler – BP 20722, F-44307 Nantes Cedex 3, France
{guillaume.richaud, hadrien.cambazard, narendra.jussien}@emn.fr

[2] Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
b.osullivan@cs.ucc.ie

Abstract. Nogood recording is a well known technique for reducing the thrashing encountered by tree search algorithms. One of the most significant disadvantages of nogood recording has been its prohibitive space complexity. In this paper we attempt to mitigate this by using an automaton to compactly represent a set of nogoods. We demonstrate how nogoods can be propagated using a known algorithm for achieving generalised arc consistency. Our experimental results on a number of benchmark problems demonstrate the utility of our approach.

1 Introduction

Nogood recording is a well known technique for reducing the level of thrashing experienced by tree search algorithms as they repeatedly rediscover the same inconsistencies. A nogood can be regarded as an assignment to a subset of the variables that cannot be extended to a solution [9]. Nogood learning was initially proposed as a Constraint Programming (CP) technique [9, 21], but without leading to significant performance improvements due to its worst-case exponential space complexity. However, it quickly became a successful technique in SAT [2]. SAT solvers seem to successfully manage the space and time requirements of nogoods. Inspired by success in SAT, several recent attempts have been made to reconsider nogoods in CP [13, 14].

There are two fundamental questions to be addressed in the context of discovering and exploiting nogoods. Firstly, *how should we compute nogoods?* Ideally one wishes to compute nogoods that rule out large parts of the future search space. Since nogoods have mostly been used to support intelligent backtracking and dynamic CSP [21], they always refer to the decision path and may be not very useful for filtering. A number of works, both in the SAT and CP community, provide some answers to this problem and we will review them in the next section. Secondly, *how should we process the nogoods we have learned?* In CP, nogoods have mostly be used to check whether the current search node can be extended to a solution or not. SAT solvers go a step further and use

[⋆] This work was supported by the Ulysses Ireland-France Travel Programme, funded in France by EGIDE (Grant Number 12476YG) and by Enterprise Ireland (Grant FR/2006/29).

a simple, but efficient, form of inference called unit propagation over the learned nogoods. Moreover, by keeping a nogood at each failure during search, one must address the problem of storing an exponential number of nogoods. The SAT community have designed clever and efficient data structures to store and propagate a very large number of clauses or nogoods. The *two watched literals* scheme [17] is one of the most successful schemes. It has also been applied in CP [13]. However the design of well suited data structures for recording nogoods in a CP context is still an open question. As a result, it is still not clear if nogood recording really pays off in CP.

We present a novel technique for storing nogoods in a compact way using an automaton. In Section 2 we recall how to compute nogoods and show how they can be propagated. In Section 3 we present our solution to store and to perform the propagation phase over a large set of tuples using an automaton. We present some preliminary experimental results in Section 4. Some concluding remarks are made in Section 5.

2 An Overview of Nogood Recording in CP

A constraint satisfaction problem (CSP) is defined by a triple $\langle X, D, C\rangle$ where $X = \{x_1, \dots, x_n\}$ is a set of variables, $D = \{D_1, \dots, D_n\}$ is the set of domains, where D_i is the domain of variable x_i and d is the maximum size of any domain. We denote by D_i^{orig} the original domain of x_i, and D_i is the current domain of x_i at a specific point of interest in the resolution process. Finally, C denotes the original set of constraints of the problem. Solving a CSP is achieved by interleaving search with propagation. Search can be regarded as the dynamic addition of constraints (decision constraints). For simplicity, we restrict ourselves to decision constraints that are of the form $x_i = a$, i.e. assignments of values to variables.

2.1 Computing Nogoods

A nogood is computed by analysing why a failure occurs during search. A nogood can be regarded as a subset of the assignments made so far that caused a failure. We introduce the basic definitions we require throughout the paper.

Definition 1 (Deduction) *A deduction $(x \neq a)$ is the determination that a value a should be removed from the domain of variable x.*

Definition 2 (Generalised Explanation) *A generalised explanation, $g\_expl(x_i \neq a)$ for the deduction $(x_i \neq a)$ is defined by two sets of constraints: $C^{'} \subseteq C$ and Δ, a set of deductions, such that $C^{'} \wedge \Delta \wedge (x_i = a)$ is globally inconsistent.*

The set Δ associated with a generalised explanation, $g\_expl(x_i \neq a)$, is denoted by $g\_expl_{\Delta}(x_i \neq a)$. Any deduction is itself, generally, due to others deductions. The inferences made during propagation can be traced back to the decision constraints added by the search algorithm. Therefore, we can compute explanations from generalised explanations. An empty Δ for a deduction $x_i \neq a$ represents the deduction that is either directly due to a decision performed on x_i such as $x_i = b$ or performed at the root node. Explaining a deduction only with respect to decisions made during search is the purpose of classical explanations.

Definition 3 (Explanation) *A (classical) explanation, $expl(x_i \neq a)$ for the deduction $x_i \neq a$ is defined by two sets of constraints: $C^{'} \subseteq C$ and DC, a set of decision constraints (assignments), such that $C^{'} \wedge DC \wedge (x_i = a)$ is globally inconsistent.*

Intelligent backtracking techniques usually store an explanation for each deduction [12]. Such explanations are computed on-the-fly by each constraint.

Example 1 (Explanations) *Let x and y be two variables such that $D_x = \{0, \dots, 6\}$ and $D_y = \{0, 2, 3, 4, 6\}$. Values 1 and 5 were removed from the domain y so an explanation is already available for these deductions. Imagine that $expl(y \neq 1) = \{x_4 = 2\}$ and $expl(y \neq 5) = \{x_0 = 3, x_8 = 1\}$ and consider the constraint $|x - y| = 2$. The value 3 of x is removed by applying the filtering algorithm on $|x - y| = 2$. A generalised explanation is simply $g\_expl_{\Delta}(x \neq 3) = \{y \neq 1, y \neq 5\}$. An explanation is, for example, $expl(x \neq 3) = expl(y \neq 1) \cup expl(y \neq 5) = \{x_4 = 2, x_0 = 3, x_8 = 1\}$.* ▲

Explanations are designed for intelligent backtracking algorithms and, therefore, they always refer to a decision path. By explaining every value that is removed, one can explain a contradiction (an empty domain D_i) by computing the union of the explanations for each value removed from D_i^{orig}. The explanation that one obtains, $expl(D_i = \emptyset) = \bigcup_{j \in D_i^{orig}} expl(x_i \neq j)$, is often called a contradiction explanation and meets exactly the classical notion of nogood, i.e. *a set of assignments that cannot be extended to a solution*.

Generalised nogoods [14] enhance the pruning power of nogoods by keeping intermediate reasons for the removal of a value instead of always projecting them onto the current decision path and *postponing* the computation of the nogood when a failure occurs. Following [12, 14], one can define a generalised nogood as follows.

Definition 4 (Generalised Nogood) *A generalised nogood is a set of constraints $C^{'}$, a set of deductions Δ and a set of decision constraints DC such that $C^{'} \wedge \Delta \wedge DC$ is globally inconsistent.*

By storing generalised explanations, one keeps in memory the logical chain of inferences made during search; in SAT this is referred to as the implication graph [4, 23]. From a contradiction due to an empty domain D_i of a variable x_i, one can compute several generalised nogoods (whereas only one nogood is available with the classical technique). The general scheme for computing a nogood[3] is given by Algorithm 1. Line 1 starts by computing the generalised nogood expressing the fact that the domain of variable x_i has been wiped-out and has raised a contradiction. Any deduction can be replaced by its generalised explanation to get a new (and maybe more informative) nogood. A generalised explanation whose Δ set is empty (line 6) is due to a decision made on that variable so we use the corresponding decision[4]. A generalised nogood is finally made of deductions as well as assignments. We can implement any SAT recording scheme by choosing the stopping criterion appropriately. For example, we can implement the Unique Implication Point [4] criterion if we wish to stop when we find a single reason that implies the conflict at current decision level.

[3] This is equivalent to the computation of a cut within the implication graph introduced in SAT. The implication graph is known in CP as a proof-tree [8].

[4] The explanation itself may be empty if the deduction is performed at the root node.

Algorithm 1 computeGeneralisedNogood(Var x_i)

```
1: GeneralisedContradictionExplanation $e \leftarrow \bigcup_{j \in D^{orig}} x_i \neq j$;
2: while stopping criterion not met do
3:     $x_k \neq k \leftarrow$ choose a deduction from $e$;
4:     if $g\_expl_{\Delta}(x_k \neq k)$ is not empty
5:         $e \leftarrow e \cup g\_expl(x_k \neq k) - \{x_k \neq k\}$;
6:     else $e \leftarrow e \cup expl(x_k \neq k) - \{x_k \neq k\}$;
7: end while
8: return e;
```

2.2 Propagating Nogoods

The propagation of nogoods is generally limited to the unit propagation approach used by SAT solvers. Consider as a literal, a variable/value pair (x_i, j). A positive literal will refer to $x_i = j$ whereas a negative literal refers to $x_i \neq j$. A positive (resp. negative) literal is said to be *satisfied* as soon as x_i is instantiated to j (resp. j removed from x_i), *falsified* in the opposite case and *free* otherwise. A generalised nogood $(\varDelta, DC)$ can be seen as a constraint, i.e. a clause over the corresponding literals $(\bigvee_{x_k \neq j \in \varDelta} x_k = j) \vee (\bigvee_{x_k = j \in DC} x_k \neq j)$, that must be satisfied in the remaining search. A nogood is free as long as two literals are free, satisfied as soon as one literal is satisfied and falsified once all literals are falsified. Moreover, the nogood is said to be *unit* when only one literal is free whereas all others are falsified. In this case, unit propagation enforces the free literal to be satisfied. The *two watched literals* scheme [4, 17] is recognised as the best way to propagate SAT clauses. We sketch this technique briefly here, since it is our baseline for nogood propagation.

The status of a nogood (free, satisfied, falsified or unit) can be determined by watching only two literals; each nogood is *watched* by two pointers on two free literals. Two lists of nogoods are, therefore, watched for each literal: the positive list, $pos\_watch(x_i, j)$ is the list of nogoods with the positive literal $x_i = j$, and the negative list, $neg\_watch(x_i, j)$ denotes the list of nogoods where $x_i \neq j$. The list $pos\_watch(x_i, j)$ is iterated once value j is removed from the domain of x_i (i.e., $x_i \neq j$) and $neg\_watch(x_i, j)$ is considered in case of an assignment (i.e. $x_i = j$). For each nogood in the list, the watched literal (now falsified) needs to be updated and several cases are considered:

1. The other watched literal is already satisfied, the pointer of the falsified literal is left unchanged;
2. Another free or satisfied literal is found and the watched list is updated accordingly;
3. Otherwise, all other literals are falsified. The nogood is unit and the other free literal is propagated. The falsified literal is left unchanged.

When using this scheme we leave the pointers to falsified literals that would remain valid upon backtracking. Indeed, the scheme ensures that as soon as a nogood is free, it is watched by two free literals. Adding a nogood dynamically at a leaf of the search tree is done by setting the pointers so that, again, the nogood will be watched correctly after backtracking. An advantage of the *watched literals* scheme is that it is well suited to applications where a large amount of memory is required for nogoods, since there is no need for complex data-structures that must be restored after backtracking.

Nogoods may still require an exponential amount of memory. It is, therefore, mandatory to forget some of the nogoods learned periodically during search. Several strategies have been introduced for forgetting nogoods such as *i-order bounded learning* [21] or *i-order relevance bounded learning* [1, 18]. Essentially these methods propose to only remember nogoods of a maximum size i or only those that are still relevant with the current decision path (that do not differ for more than i elements from the decision path). The space complexity of the previous schemes is $O(n \times d^i)$. However this tradeoff does not generally pay off [13, 15] and lots of nogoods may need to be recorded to make the learning worthwhile.

Efficient propagation schemes and optimised space management of nogoods are, therefore, the limiting factors of nogood recording techniques. To tackle the space bottleneck of nogoods, in this paper we propose to store them in a compiled form such as an automaton. Our assumption is that nogoods may share a lot of literals when they are learned in the same sub-tree. Based on the automaton representation we show that propagation can be performed efficiently. We discuss both of these issues below.

3 Encoding Nogoods using Automata

Generalised nogoods can be regarded as tuples. Specifically, since we consider finite domains, a deduction $(x_i \neq v_j)$ on domain D_i can be seen as $\bigvee_{v_k \in D_i \setminus \{v_j\}} (x_i = v_k)$. A set of tuples over n variables can be encoded in an acyclic automaton with $l = (n+1)$ layers corresponding to each variable and a final state F. A deterministic finite automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states and $q_0 \in Q$ is a starting state. The alphabet Σ corresponds to the union of all domains of the variables and Σ^* to the set of all words. Each variable x_i is associated with the i^{th} layer of the automaton and outgoing transitions of nodes belonging to layer i are labelled with values of the domain D_i^{orig}. δ is a transition function from $Q \times \Sigma \mapsto Q$ and $\delta(q, val)$ denotes the state reached by applying the transition val in state q and the pair (q, val) denotes the corresponding edge. δ^* extends δ such that

$$\delta^*(q,w) = \begin{cases} \delta^*(\delta(q,x),y) & \text{if } w = xy \text{ with } x \in \Sigma \text{ and } y \in \Sigma^*; \\ \delta(q,w) & \text{if } w \in \Sigma. \end{cases}$$

We denote by $\gamma(q_1, q_2)$ the transition values that permit moves from state q_1 to state q_2. We will denote by $|A|$ the number of states of the automaton A and by $|A_i|$ the number of states of the i^{th} layer. An example of such an automaton is given in Figure 1.

This representation has already been used in the context of constraint satisfaction problems [22]. An automaton is a generic way of representing a set of tuples and, therefore, to define a constraint in an extensional manner. For a given set, S, of tuples over a finite sequence of variables X, we will refer to:

- A the automaton recognising the feasible tuples corresponding to S. That is to say, $\mathcal{L}(A) = \{w \in \Sigma^* / \delta^*(q_0, w) = F\} = S$.
- $\overline{A}$ the automaton recognising the infeasible tuples corresponding to S. In other words, $\mathcal{L}(\overline{A}) = \{w \in \Sigma^l / w \notin \mathcal{L}(A)\}$ where Σ^l denotes the words whose length is l.

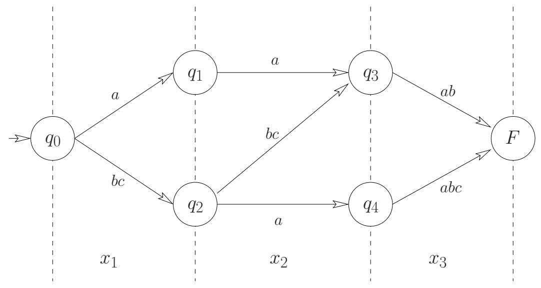


Fig. 1. An example of an automaton for three variables x_1, x_2, x_3 of domain $\{a, b, c\}$ encoding the tuples (a,a,a), (a,a,b), (b,b,a), (b,b,b), (b,c,a), (b,c,b), etc.

- $A(X)$ the automaton projected onto the current state of the domains of variables X *i.e.* that all edges (q, j) for a state q located on layer i such that $j \notin D_i$ are removed.

When the automaton is minimised (for a given ordering of the variables) it is unique, i.e. it has a canonical form. An automaton is minimal if there are no equivalent states. Two states are equivalent iff they define the same right language: $\overrightarrow{\mathcal{L}}$, i.e. they have the same set of strings that enable us to reach the final state. As we consider a layered automaton, we can efficiently minimise the automaton using bottom-to-top methods based on a recursive definition of right language of a state:

$$\overrightarrow{\mathcal{L}}(q) = \{a\overrightarrow{\mathcal{L}}(\delta(q,a))/a \in \Sigma \wedge \delta(q,a) \neq \bot\} \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F; \\ \emptyset & \text{otherwise.} \end{cases}$$

Interestingly, in a minimal automaton, encoding the infeasible or feasible tuples does not matter in terms of the size of the automaton. One can easily prove that the numbers of states of A and $\overline{A}$ differs by at most l states.

Property 1 *If A and $\overline{A}$ are minimal then $abs(|A| - |\overline{A}|) < l$.*

Proof. (Sketch) One can show how to build $\overline{A}$ from A (see Figure 2). First, change the final state of A into a garbage state so that all valid tuples of A become forbidden. Second, invalid tuples of A have to be recognised and all missing transitions (those going implicitly towards the garbage state) have to be added (bold edges on Figure 2). At most $(l-1)$ states are removed (dashed edges on Figure 2). Indeed a state is removed if all its transitions lead to the old final state, and only one state per layer may have such a property (otherwise the two states would have been equivalent). Again, at most one state is added per layer (because, again, of minimality only one state may have all its outgoing transitions leading to the new final state). □

Considering a set of nogoods S (infeasible tuples), we choose to maintain the automaton $\overline{A}$ corresponding, therefore, to the set of feasible tuples. Adding a nogood within such an automaton means removing the corresponding word from the language recognised by the automaton. While this does not really matter for the automaton's size, it is easier to reason on $\overline{A}$ when propagating the automaton.

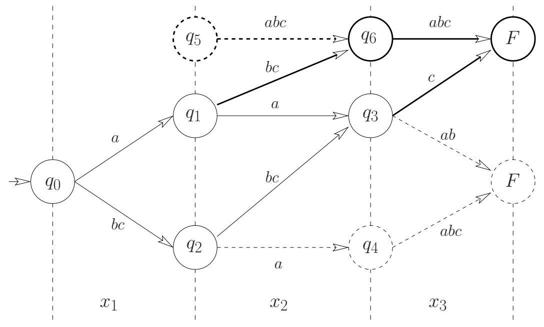


Fig. 2. Switching from A (normal and dashed edges) to $\overline{A}$ (normal and bold edges).

3.1 Incremental Minimisation of the Automaton

We briefly describe two strategies for incrementally minimising the automaton. The goal is to incrementally maintain the automaton of feasible tuples. We must be able to add nogoods (remove the corresponding word from the language recognised by the automaton) incrementally as we discover new ones [11].

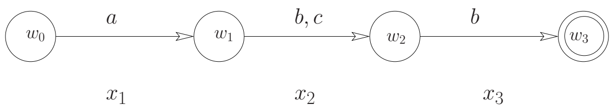
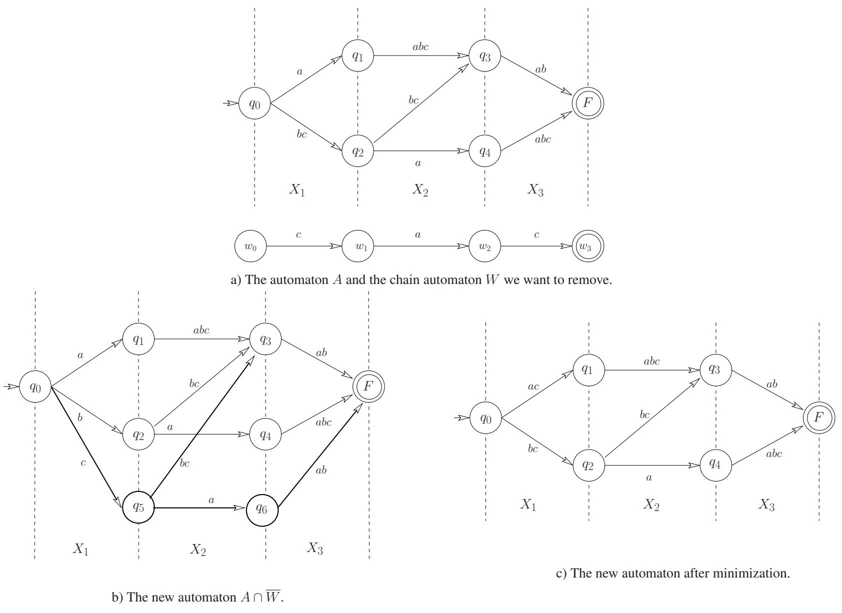


Fig. 3. Chain automaton recognising the generalised nogood $(x_1 = a) \wedge (x_2 \neq a) \wedge (x_3 = b)$ with $\Sigma = \{a, b, c\}$.

We consider a finite Σ so each deduction $(x \neq v)$ of the nogood can be replaced by $\{x \in \Sigma \backslash v\}$. We denote by w the word corresponding to the nogood to be removed from the allowed tuples of $\mathcal{L}(A)$. Removing w from $\mathcal{L}(A)$ involves building a new automaton $A \cap \overline{W}$ with W the chain-automaton recognising w (see Figure 3). W is built using the same variable order of A such $\delta^*(w_0, w)$ is the final state.

The algorithm proceeds in two steps (depicted Figure 4). Firstly, we compute $A \cap \overline{W}$: the main differences with other methods used to incrementally construct minimal acyclic automata is that we try to remove a string instead of adding it and that a generalised nogood can represent more than one string. Secondly, we incrementally minimise the new automaton by taking into account the new added states; using the fact that our automaton is layered we can minimise it efficiently. The time complexity of the removal and minimisation is $O(|W| + |\Sigma| \times |W|)$.

Adding a nogood w can add at most $|w|$ states to the automaton even if no minimisation occurs (see Step b of Figure 4). This is, however, not true for generalised nogoods. We find the incremental compilation of nogoods difficult for the following three reasons. Firstly, in the case of generalised nogoods, the automaton can be larger (in number of states) than the sum of the number of states of chain automata corresponding to the nogoods. This is due to the fact that a generalised nogood represents several tuples. A chain automaton is already a kind of compact representation. Moreover this



q_0
q_1
q_2
q_3
q_4
q_5
q_6
F
a
bc
abc
ab
b
c
ac
X_1
X_2
X_3
w_0
w_1
w_2
w_3
a) The automaton A and the chain automaton W we want to remove.
b) The new automaton $A \cap \overline{W}$.
c) The new automaton after minimization.

Fig. 4. The minimisation process.

behaviour is difficult to predict as it is hard to predict the size of the automaton for a given language. In the worst case, adding one generalised nogood to the automaton can add $\sum_{k=2}^{l-1} |A_k|$ new states. Secondly, the size of the automaton is related to the order of the variables. As tuples are discovered during search, the dynamic computation of the automaton would imply we re-order it dynamically. Thirdly, by adding the nogoods dynamically one by one in the order in which failures are encountered in search, the size of the automaton may increase quickly and will only decrease when enough nogoods have been learned so that they share a sufficient number of assignments.

We are currently investigating how to delay the compilation of nogoods in the hope that, as the nogoods are known, it would be possible to: select a subset of nogoods that may give a compact automaton; find a good order on the variables by applying some heuristics (based on similar ideas than for ROBDD [5]) before compilation; optimise the order in which nogoods are added to the automaton during the incremental minimisation phase to avoid large intermediate sizes [16].

3.2 The Filtering Algorithm

Pesant [19] provides a filtering algorithm for a global constraint defined by a regular language. We use this algorithm to enforce arc-consistency using our automaton A. The idea behind the algorithm is to maintain the set Q_{ij} of states acting as supports for each variable-value pair (x_i, v_j). A state q of the ith layer is considered as a support of (x_i, v_j) as long as there exists a path from q_0 to q and from $\delta(q, v_j)$ to F in A. Once Q_{ij} is empty, value j is removed from the domain of variable x_i.

In Figure 1, for example, we have $Q_{1a} = Q_{1b} = Q_{1c} = \{q_0\}$, $Q_{2a} = \{q_1, q_2\}$, $Q_{2b} = Q_{2c} = \{q_2\}$, $Q_{3a} = Q_{3b} = \{q_3, q_4\}$ and $Q_{3c} = \{q_4\}$. Incremental propagation is performed by storing in a "backtrackable" data-structure the incoming and outgoing edges of each node as well as their in and out-degree. Each time a value j is removed from the domain of variable i, the degree of the states within Q_{ij} are decremented accordingly. If some degree reaches zero, this information is propagated to all connected nodes (predecessors if the out-degree is null and successors in case of the in-degree) by decrementing their degree and maintaining the Q_{ij} lists accordingly.

In the example in Figure 1, if values a and b are removed from the domain of x_3, the out-degree of q_3 falls to zero, so its ingoing edges are considered. The states q_1 and q_2 are removed from Q_{2a}, Q_{2b} and Q_{2c} while iterating over the ingoing edges. As Q_{2b} and Q_{2c} are updated to $\emptyset$, values b and c are removed from x_2. Moreover, the out-degrees of q_1 and q_2 are decremented and the process continues as the degree of q_1 reaches zero so that value a is finally removed from x_1.

Explaining Automaton-based Filtering. As stated in Section 2.1, each filtering algorithm has to be *explained* in order to be able to generate generalised nogoods. Each time a value is removed, a generalised explanation must be associated with the deduction. It is, therefore, mandatory to explain the pruning that comes from the nogoods compiled in the automaton. Explaining the pruning of the automaton is done by, firstly, explaining why a state cannot reach F (Algorithm 2) and, secondly, explaining why a state can not be reached from q_0 (Algorithm 3). An explanation $expl(q)$ and a back-

Algorithm 2 explainOut(State q, int i)

```
1: Explanation $e \leftarrow \emptyset$;
2: if is_explained(q) is false then
3:     for all $j$ such that $\delta(q, j) \neq null$ do
4:         if $j \in D_i$ then $e \leftarrow e \cup expl(\delta(q, j))$;
5:         else $e \leftarrow e \cup expl(x_i \neq j)$;
6:     end for
7:     is_explained(q) ← true;
8:     $expl(q) \leftarrow e$;
9: end if
```

Algorithm 3 explainIn(State q, int i)

```
1: Explanation $e \leftarrow \emptyset$;
2: if is_explained(q) is false then
3:     for all $(p, j)$ such that $\delta(p, j) == q$ do
4:         if $j \in D_{i-1}$ then $e \leftarrow e \cup expl(p)$;
5:         else $e \leftarrow e \cup expl(x_{i-1} \neq j)$;
6:     end for
7:     is_explained(q) ← true;
8:     $expl(q) \leftarrow e$;
9: end if
```

trackable boolean, $is\_explained(q)$, are associated with each state q of the original automaton. $expl(q)$ records why q is invalid, i.e why it cannot be on a path from q_0 to F. $is\_explained(q)$ is true if the invalidity of q has already been explained and a valid $expl(q)$ is available in the current branch of the tree. Since many explanations exist it is mandatory to avoid overriding an existing valid explanation, because the explanation itself is not restorable upon backtracking.

A value j from a variable x_i is pruned because Q_{ij} is empty. We can explain the pruning because for each state q that was part of the original list of supports of (x_i, v_j) (denoted $Qinit_{ij}$), either q is itself invalid or $\delta(q, j)$ is invalid (Algorithm 4).

Algorithm 4 prune(int i, int j)

```
1: Explanation $e \leftarrow \emptyset$;
2: for all $q$ in $Qinit_{ij}$ do
3:     if is_explained(q) then $e \leftarrow e \cup expl(q)$;
4:     else $e \leftarrow e \cup expl(\delta(q, j))$;
5: end for
6: remove value $j$ from $x_i$ due to $e$;
```

$expl(q)$ is computed for each state q in the following way. Firstly, to explain why a state q_k at layer i cannot be reached from q_0, we divide its predecessors into two sets $rpred$ and $\overline{rpred}$. The predecessors $rpred$, that can be reached from q_0, and those, $\overline{rpred}$, that are unreachable. For each predecessor p of q_k, either it belongs to $\overline{rpred}$ and we use the explanation $expl(p)$ attached to p, or it belongs to $rpred$ and the values of transitions leading to q_k from p ($\gamma(p, q_k)$) have been removed from the domain of x_{i-1}. Algorithm 3 is called each time the in-degree of q_k reaches zero and computes $expl(q_k)$:

$$expl(q_k) = expl(q_0 \not\Rightarrow q_k) = \bigcup_{p \in rpred} expl(x_{i-1} \neq \gamma(p, q_k)) \cup \bigcup_{p \in \overline{rpred}} expl(p).$$

Secondly, in a similar way, the state q_k cannot reach F because either its successor cannot reach F or the value leading to a state that could reach F is missing. Algorithm 2 is called each time the out-degree of q_k reaches zero and computes $expl(q_k)$:

$$expl(q_k) = expl(q_k \not\Rightarrow F) = \bigcup_{s \in rsucc} expl(x_i \neq \gamma(q_k, s)) \cup \bigcup_{s \in \overline{rsucc}} expl(s).$$

Lightweight Filtering Algorithms. The aim of the automaton is to compile large sets of nogoods and, therefore, to be able to mitigate the large space consumption of classical approaches. The incremental propagation algorithm is, in a sense, very greedy in memory as it needs two doubly-linked lists (incoming and outgoing arcs) and two integers (in-degree and out-degree) per state that are restorable upon backtracking. It also uses a backtrackable list Q_{ij} of states per variable-value pair. First, we give up maintaining doubly-linked lists for ingoing and outgoing edges. If the number of outgoing edges is bounded by the alphabet size (the maximum domain size), the number of ingoing edges can be equal to the number of states of the previous layer which seems unreasonable in our case. This algorithm is denoted by $Aut0$ in the following. Moreover, we investigate the following tradeoff which looses the constant time update at each variable-value removal: firstly, explained by Pesant [19], one does not need all the state-supports and only one can be kept in memory; secondly, one does not really need the exact degree of each state but only whether the degree is null or not.

One strength of watched literals precisely lies in the fact that nothing needs to be restored upon backtracking. We tried, based on this principle, to spare memory by keeping an outgoing and ingoing edge per state that are updated only when the edge is lost instead of storing the degree. A valid edge at depth k in the tree search is also valid at depths less than k. The filtering based on $Aut0$ with the previous improvement is denoted $Aut1$. Finally, we store only one support-state for each value (x_i, v_j). When this support become invalid, we look for an other one among edges of $Q_{init_{ij}}$. $Aut1$ combined to this improvement is called $Aut2$.

4 Empirical Evaluation

We present experiments that study two aspects of the problem studied in this paper. In Section 4.1 we investigate the value of storing a large table of tuples in an automaton to perform filtering compared to generalised arc-consistency [3]. In Section 4.2 we report our experience of nogood recording with watched literals. For the reasons presented in Section 3.1, the dynamic compilation of nogoods was far too costly to be competitive. Experiments for the automaton remain to be done once the questions raised in Section 3.1 have been addressed. Crossword puzzles and RLFAP are our benchmark problems. All experiments are performed on a Pentium 4 3GHz with 1 GB of RAM under Linux with the choco constraint solver (*choco-solver.net*).

4.1 The Automaton: Storing and Filtering

Crossword puzzles problems involve filling a given grid using words from a reference dictionary such that each word is used at most once. Our interest here is that constraints have to store large tables of tuples corresponding to allowed words of the dictionary.

A variable x_i with domain $D(x_i) = \{a, b, \dots, z\}$ is associated with each free square of the puzzle. A constraint is stated per word, i.e., per contiguous sequence of letters in the puzzle. The allowed tuples of the constraint are defined by all words of the corresponding length from a reference dictionary. A word can only be used once in the puzzle so a not-equal constraint is also added between any pairs of words with the same size. We studied two approaches to enforcing GAC on the problem:

1. The propagation scheme described above. Each dictionary of size k (all words of size k) is compiled within a minimal automaton called $auto_k$ (Q^k_{ij} denote the set Q_{ij} for $auto_k$).
2. The GAC schema introduced by [3][5]. A direct access to the supports of each variable-value pair is given within a shared data-structure among constraints. Linked-lists of words of size k that have a letter l at a given position p are stored in a three dimensional array called `supports[l][p][k]`. GAC is then achieved with a GAC2001 algorithm by storing the current support (an integer restorable upon backtracking denotes the index of the word in the `supports` data structure).

We considered the benchmark of [7] which is made of instances from size 5×5 to 23×23 and comes from the Herald Tribune Crosswords. We use the dictionary *words* that collects 45000 words. Table 1 summarises the results (time limit is set to 1 hour).

The initial propagation of the automaton is costly (initialisation of the Q_{ij} lists). GAC2001 is, therefore, faster on instances that are solved in a few nodes. However,

[5] Multidirectionality is not implemented in our GAC schema.

Table 1. Automaton and GAC filtering for crossword puzzles.

	Mac-Aut0		Mac-Aut1		Mac-Aut2		Mac-GAC	
Instances	**Time(s)**	**Node**	**Time(s)**	**Node**	**Time(s)**	**Node**	**Time(s)**	**Node**
05.01(dico:words)	**0,2**	30	0,2	30	0,3	30	0,3	30
15.01(dico:words)	1,3	75	1,3	75	1,3	75	**0,9**	75
15.02(dico:words)	**12,1**	872	13,7	872	16,6	872	25,5	872
15.07(dico:words)	**321,2**	22859	366,4	22859	554,1	22859	923,4	22859
19.02(dico:words)	**83,8**	17511	95,2	17511	214,6	17511	253,2	17511
19.05(dico:words)	$>$ 1h	**1213314**	$>$ 1h	1066428	$>$ 1h	728579	$>$ 1h	500871
21.03(dico:words)	**63,6**	13017	73,2	13017	203,5	13017	248,7	13017
21.06(dico:words)	$>$ 1h	**494848**	$>$ 1h	416718	$>$ 1h	261107	$>$ 1h	131916
21.07(dico:words)	**20,2**	1825	23,4	1825	34,5	1825	49,5	1825
23.07(dico:words)	$>$ 1h	**256456**	$>$ 1h	227168	$>$ 1h	118092	$>$ 1h	102668

on hard instances, the automaton tends to be between two and three times faster than GAC2001. This is due to the fact that words are a "structured" set of tuples (that share a lot assignments), i.e $|Q^k_{pl}| < |$`supports[l][p][k]`$|$. The result is that the function `seekNextSupport` which is the basis of any GAC algorithm is faster on Q^k_{pl} than on `supports[l][p][k]`.

Table 2. Average memory consumption (in Mbytes) for the four approaches.

instance	$Aut0$	$Aut1$	$Aut2$	GAC
15.07(dico:words)	31.8	21.1	16.1	19.7
19.02(dico:words)	37.2	25.2	17.6	19.1
21.03(dico:words)	51.6	33.9	22.2	23.7

In terms of memory, the automaton is more compact (1,70 Mbytes) than the `supports[l][p][k]` data structure (3,98 Mbytes) for storing all words of size 8 of the dictionary *words*. However, the data structure needed to filter the automaton consumes more memory than the GAC. Every 500 backtracks we measure the amount of memory used for the four approaches and report the average (Table 2). Among the three versions, the best compromise for space and time requirements seems to be $Aut1$. Notice that $Aut2$ is a little faster than GAC but requires less memory.

4.2 Nogood Recording

We studied the following three approaches on the Crossword puzzles problems and RLFAP (Radio Link Frequency Allocation Problems): MAC-CBJ [20] is an intelligent backtracking technique that involves in backtracking to the latest decision involved in the conflict when a failure occurs; MAC-CBJ + S is MAC-CBJ combined to standard nogood propagated by watched literals; MAC-CBJ + G is MAC-CBJ combined to generalised nogoods propagated by watched literals. The variable ordering heuristic used was min(dom/deg).

Crosswords puzzles. We again study crossword puzzles. The time limit was set up to 2 hours and $aut1$ is used for the constraints stated per word. Results are reported

Table 3. The use of watched literals implies a little overhead so MAC-CBJ remains faster on easy instances. However, we were able to find the results reported in [14], so generalised nogoods do effectively pay off on this problem. The next step will be to see the utility of the automaton for storing such nogoods.

Table 3. Nogood recording for crossword puzzles.

	MAC-CBJ		**MAC-CBJ + S**		**MAC-CBJ + G**	
	tps (s)	**node**	**tps (s)**	**node**	**tps (s)**	**node**
15.02(dico:words)	**29,7**	314	46,2	314	43,8	**303**
15.07(dico:words)	**841,6**	18182	1255,6	17918	894,3	**11172**
19.02(dico:words)	**10,8**	264	15,6	264	14,1	**219**
19.05(dico:words)	110,6	2104	64,2	1182	**47,9**	**727**
21.03(dico:words)	**11,8**	292	18,4	292	18,3	**281**
21.06(dico:words)	177,0	2707	265,8	2677	**164,8**	**1879**
21.07(dico:words)	**44,9**	1168	72,6	1157	63,5	**957**
21.04(dico:words)	> 2h	87677	> 2h	73342	> 2h	47527
23.07(dico:words)	**45,3**	954	74,7	892	61,7	**584**
21.05(dico:words)	> 2h	176226	> 2h	102057	**722,8**	9407
21.10(dico:words)	> 2h	74718	> 2h	47346	> 2h	43440
15.04(dico:words)	> 2h	171029	> 2h	105069	**1268,6**	**16315**
15.06(dico:words)	> 2h	150552	> 2h	94936	**5529,1**	**50554**
15.10(dico:words)	> 2h	153260	> 2h	99929	**1634,6**	**14153**
19.03(dico:words)	> 2h	123770	> 2h	88461	> 2h	58632
19.04(dico:words)	> 2h	303559	> 2h	222910	**55,2**	**2294**
19.07(dico:words)	> 2h	462297	> 2h	289966	**131,9**	**4312**
21.01(dico:words)	> 2h	86857	> 2h	53418	> 2h	36336
23.03(dico:words)	> 2h	80810	> 2h	50998	> 2h	42919
23.04(dico:words)	> 2h	74641	> 2h	37265	> 2h	23723
23.05(dico:words)	**11,7**	239	17,1	254	15,7	**226**

Radio Link Frequency Allocation Problems. Our second experiment is based on real world frequency allocation problems coming from the FullRLFAP archive [6]. The problem involves finding frequencies (f_i) for different channels of communication so that interferences are minimised. We followed the approach described in [10] to generate hard satisfaction instances. Therefore, `scenXX-wY-fZ` corresponds to the original instance `scenXX` where constraints with a weight greater than `Y` are removed, as well as the `Z` highest frequencies. Generalised nogoods are much more effective on those problems again (Table 4) however the results of [10], using a different conflict-based heuristic to ours, remain better.

Figure 5 shows the size of the automaton after the addition of each of the first 300 generalised nogoods taken from `scen03-05-11`. The basic compilation approach is dyn in which nogoods are compiled as they are discovered, using an ordering for the variables in the automaton based on the variable ordering from the nogoods themselves. We study two independent enhancements to the basic scheme. Firstly, denoted by $+ls$, we use a local search to optimise the variable ordering before compiling a set of nogoods; for example, when compiling k nogoods we try to find an ordering for the automaton that minimises its size having compiled the k nogoods. Secondly, denoted by $+s$, we compile the k nogoods in lexicographically order (instead of compiling them in the order given by the failures during search). Notice that $+ls$ affects the final size

Table 4. Nogood recording for RLFAP problems.

	MAC-Cbj		**MAC-Cbj + S**		**MAC-Cbj + G**	
	tps (s)	**node**	**tps (s)**	**node**	**tps (s)**	**node**
scen02-05-24	**0,3**	104	0,9	104	0,4	104
scen02-05-25	**3,0**	610	5,2	610	3,1	**360**
scen03-05-10	1659,2	572507	> 2h	343927	**123,8**	**11575**
scen03-05-11	> 2h	3506415	> 2h	776095	> 2h	155008
scen11-05-00	6,4	1207	8,3	1207	**3,8**	**622**
scen06-02-00	73,9	68669	164,2	61866	**5,3**	**1854**
scen07-01-04	**0,1**	202	0,2	202	0,2	**201**
scen07-01-05	**0**	26	0,1	26	0,1	26
graph08-05-10	> 2h	1722485	> 2h	491970	> 2h	175079
graph08-05-11	> 2h	1300390	> 2h	494286	**46,6**	**6906**

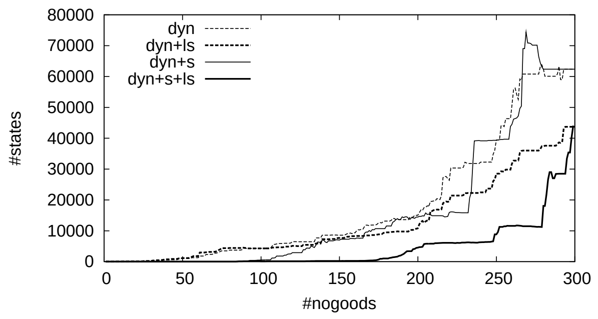


Fig. 5. Size of the automaton (#states) after the addition of each nogoods for scen03-05-11.

of the automaton, since it affects the variable ordering, whereas $+s$ affects only its intermediate size.

In Figure 5 we see two pairs of curves: one pair corresponds to the nogood-based variable ordering, while the other used local search. Within each pair of curves, while each one converges on the same size automaton having compiled all the nogoods we considered, the intermediate size is determined by the order in which the nogoods were compiled. Interestingly, a lexicographic ordering of the nogoods does not ensure that the automaton is more compact than the ordering based on how the nogoods themselves were discovered. Clearly, the variable ordering in the automaton is critical to ensure an overall compact representation, but the order in which nogoods are incrementally compiled is also important in order to avoid a large intermediate automaton.

5 Conclusion

This paper investigates a novel approach to storing and propagating nogoods. The approach uses an automaton to overcome the exponential memory requirements of nogood recording. We demonstrate the advantages and limitations of the approach. We show that the dynamic compilation of nogoods is certainly very difficult to achieve in practice but we show interesting computational results using an automaton to achieve arc-consistency over large and structured tables of tuples.

References

1. R. J. Bayardo and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI-1996*, pages 298–304, 1996.
2. R. J. Bayardo and R. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Proceedings CP 1996*, pages 46–60, 1996.
3. C. Bessière and J.-C Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI'97*, pages 398–404, 1997.
4. L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. Technical Report MSR-TR-2005-124, 2005.
5. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
6. B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.
7. X. Chen and P. Beek. Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research*, 14:53–81, 2001.
8. R. Debruyne and al. Correctness of constraint retraction algorithms. In *FLAIRS'03*, pages 172–176, 2003.
9. R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
10. C. Lecoutre F. Boussemart, F. Hemery and L. Sais. Boosting systematic search by weighting constraints. In *ECAI'04*, pages 482–486, 2004.
11. B. Watson J. Daciuk, S. Mihov and R. Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, 2000.
12. N. Jussien and P. Boizumault. Dynamic backtracking with constraint propagation – application to static and dynamic CSPs. In *CP Workshop: Theory and Practice of Dynamic Constraint Satisfaction*, 1997.
13. G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In *Proceedings CP 2003*, pages 873–877, 2003.
14. G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *National Conference on Artificial Intelligence (AAAI-2005)*, pages 390–396, 2005.
15. I. Lynce and J. Marques-Silva. The effect of nogood recording in MAC-CBJ SAT algorithms. Technical Report RT/04/2002., 2002.
16. S. Mihov. Direct building of minimal automaton for given list. In *Annuaire de l'Université de Sofia St. Kl. Ohridski*, volume 91. Sofia, Bulgaria, 1998.
17. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of DAC'01*, 2001.
18. S. Ouis, N. Jussien, and P. Boizumault. k-relevant explanations for constraint programming. In *FLAIRS'03*, pages 192–196, St. Augustine, Florida, USA, 2003.
19. G. Pesant. A regular language membership constraint for finite sequences of variables. In *CP 2004*, volume LNCS 3258, 2004.
20. P. Prosser. MAC-CBJ: maintaining arc consistency with conflict-directed backjumping. Technical Report /95/177, University of Strathclyde, 1995.
21. T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problem. *IJAIT*, 3(2):187–207, 1994.
22. N.R. Vempaty. Solving constraint satisfaction problems using finite state automata. In *AAAI*, pages 453–458, 1992.
23. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.